EMBEDDABLE SOURCE CODE EDITOR

FIELD OF THE INVENTION

The present invention relates generally to editors for source code including source code for mark-up languages, and more particularly to a language-based editing architecture that can simplify reading and writing source code, while providing an interfacing editor that can share information with other tools and be packaged for re-use.

BACKGROUND OF THE INVENTION

Source code plays a major role in most software engineering environments, yet it has always been difficult for software engineers to interface with such code. As used herein, the term source code shall be understood to include code representing software that includes mark-up language such as HTML and XML, in addition to representing programming language software.

A common problem programmers encounter when entering and displaying source code is that prior art program editors are language-sensitive and take into account each keystroke entry that is made, but do not account for when in time such keystroke was made. As will be now be described, this can result in confusing any language-based display (or prettyprinting) and can result in the loss of any previous linguistic analysis of the program structure being entered.

Fig. 1 depicts a typical computing system 10 with which a programmer might enter source code. System 10 will include a computer system 20 having at least one CPU 30 and memory (MEM) 40 contained therein that will typically include non-volatile memory and volatile memory. A portion of the volatile memory is typically used as a text buffer 50. A programmer enters source code into computer system 20 typically using a keyboard 60, often augmented by use of a mouse or trackball 70. Text buffer 50 is sometimes referred to as a data

model, and is where input data is stored while being entered from the keyboard or other input device by the programmer.

A view function carried out by computer system 20 displays or paints video on a monitor 80 that is shown here displaying programmer-input text 90 followed by a cursor 100. The view function, which may be carried out by CPU 30 executing code stored in memory 40, typically includes style information for a particular language type (e.g., comment, string literal, tag) such as font, color, etc. The view function looks at the contents of buffer memory 40 and applies what is believed, rightly or wrongly, to be the correct stylistic rules in creating display 90 on monitor 80.

A control function is typically the editor function or editor kit, and is code that, upon execution by CPU 30, implements behavior as to what next should be done with respect to language and desired behavior. Exemplary such editors, referred to herein as a parent editor, may be found in applicant's U.S. patent no. 5,752,058 "System and Method for Inter-token Whitespace Representation and Textual Editing Behavior in a Program Editor" (1998), and his U.S. patent no. 5,748,975 "System and Method for Textual Editing of Structurally-Represented Computer Programs With On-the-fly Typographical Display" (1998), both patents being assigned to the present assignee herein. Applicant refers to and incorporates each said patent by reference herein. Not every keystroke entry will necessarily change the model or buffer contents, e.g., a mere cursor movement might not change the buffer contents. However keystroke entries that the controller or editor decide should change the model or buffer contents will result in painting a new view on display 80 as the buffer contents change.

Referring now to Fig. 2A, assume that the programmer enters the following keystrokes:

S = E + "}" ;

The above entry is shown as display 90 in Fig. 1 and as contents of the text buffer 50 in Fig. 2A. A prior art source code editing system 10 recognizes language features of entered text by performing pattern matching or lexical (language) analysis upon keystrokes. Referring to the text buffer (also called a character array) contents shown in Fig. 2A, a pattern matched or lexical analyzer recognizes the group of tokens 110, namely "}", as a string literal. Having properly recognized string literal 110 from among the remaining source code language contents of text buffer 50, prior art system 10 can readily apply special typographical attributes such as special coloring when displaying this language on monitor 80.

Assume now that the programmer wishes to amend this line so it will eventually read as follows:

S = "{" + E + "}";

To carry out the above change, the programmer would place cursor 100 on display 90 before the identifier E and type a double quote " using keyboard 60. The " (double quote) is the first character to be entered as part of the changes about to be entered by the programmer, and at this juncture the state of the text buffer will be as shown in Fig. 2B. However because prior art editing systems do not take into account when in time a triggering keystroke is made, e.g., here a ", which signifies start of a string literal, system 10 does not know how to interpret the present state of buffer 50 properly, e.g., as intended by the programmer.

Thus, as shown in Fig. 2B, since the system pattern matcher or lexical analyzer re-analyzes the text buffer contents after each keystroke, it will erroneously assume that portion 120 of the buffer contents, name "E + " is a string literal comprising E +. Accordingly as soon as the programmer entered the triggering keystroke ", the erroneous assumption(relative to the programmer's intent) as to E + being a string literal is made and typographical attributes on display 90 are made. Since it no longer appears to be preceded by an opening ", line portion 130 } will be interpreted wrongly as code, rather than contents of a string literal

as intended by the programmer. But since E was not intended by the programmer to be a string literal, the color and any other typographical attributes changes made on display 90 will be wrong, and will most likely distract the programmer entering the keystrokes. Just as E+ finds itself temporarily surrounded by double quotes, it is now interpreted as a string literal, and buffer contents 140 will be treated by system 10's lexical analyzer as being an invalid lexeme. More specifically, system 10 will treat buffer contents 140 as being the start of a string literal that is lacking a closing double quote.

In the above example, as the programmer adds keystrokes to arrive at the desired S = "{" + E + "}"; entry, keystroke-by-keystroke the prior art editor will interpret what appear to be string literals and what appear to be source code, and will change display 90. However until all keystrokes necessary to enter the desired S = "{" + E + "}"; have been entered, computer system 10 will misinterpret what is being entered, for example treating all entries after an initial double quote " as part of a string literal, until the closing double quote " has been entered. As noted, the constant changing appearance of display 90, keystroke-by-keystroke, including color and other typographical changes due to what system 10 incorrectly believes (relative to what the programmer intended) to be string literals can be very confusing to the programmer entering the source code.

Central to any specialized editing tool is an internal representation for source code. Prior art representations range from text-based editing to language structure-based editing. Text-based editors employ an ad hoc matching to recognize certain keystrokes, and employ a rather simple user-model having minimal structure and no language-based (or linguistic) support, which can be desirable to programers. Language structure-based editors typically use text escapes and have a rather rich but highly structured model that can be advantageous for interfacing with other services. Language structure editors are so-called in that they have internal representations that are closely related to tree and graph structures used by compilers and other tools. Unfortunately, purely

lexical-based architecture are not optimum in that their naive user-model would suffer many of the ills associated with tree-oriented editors.

A typical compiler analyzes textual programs in phrases in the following manner:

lexical analysis → parsing → static semantic analysis

(which nomenclature corresponds approximately to types 3, 2, 1 in the grammar hierarchy put forth by Chomsky), and uses a corresponding type of analyzer for each analysis phase. Programming languages are often designed around such grammatical decomposition, and batch-oriented compilers can benefit from the simplicity and formal foundations of separate phases. Ranging from simple to more complex user models, still with minimal structure (better for users, but less so services) a spectrum of editor design choices would range from pure text, to text plus regular expression to editors using lexical tokens. At the opposite extreme, representing high structure (better for services but less so for users), and rich to less rich representations are found fully-attributed syntax trees, syntax trees with fuzzy attribution, and parse trees. In the middle of the spectrum might be found editors based upon lexical tokens and fuzzy parsing. Fuzzy parsing is a partial syntactic analysis that recognizes only certain features of the context-free syntax (e.g., nested parenthesis, or context-dependent categorization of identifiers into function and variable names), and partial semantic attribution useable for computing limited amounts of semantic context. Partial analyses can be simpler to implement, and are also more forgiving of inconsistencies in the representation.

The above three analytical phases differ concerning the scope of cause and effect. Static semantic analysis (closely related to Chromsky's context-sensitive syntax) depends at each point in a program potentially upon the entire program. Parsing (context-free syntax) depends only on the enclosing phrase, but assumes the program is well-formed. Lexical analysis (regular syntax) depends only upon adjacent tokens, which can make lexical analysis very suitable for the inner loop of an editor.

So-called structure editors use internal representations that are closely related to the tree and graph structures used by compilers and other tools. Structure editors can greatly simplify some kinds of language-oriented services, but impose the requirement that the programmer edit using structural rather than textual commands. From a tool integration perspective, any advantages of complete linguistic analysis are offset by the fragility, in the presence of user editing, and the context-dependency (the meaning of code in many languages depends potentially on all the other code with which it will run) associated with structure editors. Such editing architecture assumes that programs are intrinsically tree structured, and should be understood and manipulated in that context. In practice, this assumption has not borne out and structure editors have not found wide acceptance.

Analyzers, builders, compilers, debuggers and other such software engineering tools generally operate over structural source code representations such as abstract syntax trees. It will be appreciated that readily integrating with such tools, requires the source code editor to share such representations. However this constraint has long presented a severe design challenge in providing a tool intended to display and permit modification ro source code in terms of text.

Programmers tend to read programs textually, with a structural understanding of the program. However the programmer's understanding can be highly variable and not necessarily based on language analysis. Programmers often exhibit deeply ingrained work habits and motor-learning that involves textual editing. It is a truism that programmers will generally accept a new tool only if it is familiar enough for immediate and comfortable use without any special training. This is a practical constraint in designing a source code editor that will truly be embraced by the programming community.

But subjective programmer constraints as noted above need not inhibit advanced functionality. For example, users experienced with simple text editors can readily transition to using word processor programs, especially as the cursor command

and text entry is already familiar to them. Although simple text editors lacking linguistic support have been used and can provide simple and familiar editing, there is no real specialization for source code. To integrate a simple text editor with software engineering tools requires complex mappings between structure and text, and what results is restrictive and often confusing in its functionality. Further, the representations are fragile, especially in that the identity of structural elements are lost during editing operations.

Some structure editors allow programmers to escape the structure by transforming selected tree regions into plain text, but usability problems persist. The complex unseen relationship between textual display and internal representation makes editing operations, including structural and text escapes, confusing and often unpredictable due to hidden states. Textual escapes are complicated with respect to parts of a program in which language-based services are provided and parts in which they are not provided. Often language-based services and tools will stop functioning until all textual regions are syntactically correct and have been promoted back into structures.

In the prior art, code-oriented text editors such as Emacs use a purely textual representation, assisted by ad-hoc regular expression matching that can recognize certain language constructs. But, by definition, the structural information computed by simple text editors is incomplete and imprecise, and such editors cannot support services that require true linguistic analysis such as advanced program typography. At best, simple text editors typically provide indentation, syntax highlighting, and navigational services that can tolerate structural inaccuracy. Although high quality, linguistically-driven typography can measurably improve the programmer's reading comprehension, such typography is often lacking in prior art source code editors, especially when encountering malformed and fragmentary program code. Although a few text editors can perform per-line lexical analysis with each keystroke, the absence of true program representation leads to confusion in the inevitable presence of mismatched string quotes and comment delimiters.

Given a choice of interfaces with source code, most software engineers would elect a tool that could display source code textually and further would permit edits or modifications to the code. If such an interface tool directed to source code could be provided, it could ease programming by providing enhanced programming services, and could also provide a tighter integration with other programming tools. But it has been difficult in the prior art to meet these two goals as they present conflicting design constraints that have long prevented specialization.

Any interactive software engineering tool that deals with programs will inevitably displays source code for a human to read and possibly modify. Although specializing the technology directed to programming languages would probably substantially enhance user productivity, prior art techniques for such displaying and modification have changed little in twenty years. This stagnation in the field of source code editing programs is in stark contrast to the dynamic changes that have occurred in word processing systems during those same twenty years.

The lack of a successful new technique for source code editing has not been for want of effort. Indeed extensive research and experimentation with numerous prototypes has been undertaken in the prior art. Unfortunately despite such efforts, there still does not exist a commercially viable, practical language-based editing system for source code. Software intended to provide such features has met with great resistance by programers who find using the system unduly cumbersome and difficult, especially when contrasted with the ease of use of simple text editors. Simply stated, prior art implementations have been found by tool builders to be overly fragile, yet highly demanding of supportive infrastructure. Prior art tools have attempted to satisfy the needs of programmers at the expense of tool builders, or have tried to satisfy the needs of tool builders at the expense of programmers, but have never successfully satisfied the constraints of both simultaneously.

Thus, there is a need for a language-based editing architecture that satisfies programmers by making reading and writing source code easier and more rewarding, and that also satisfies tool builders by providing an editor that can reliably share information with other tools, that can act as a user interface to the source code, and that can be packaged for re-use, e.g., in a format that is portable, highly configurable, and embeddable. Such editing architecture and editor should provide language-based editing services that are layered carefully onto basic text editing behavior skills, without imposing substantial restrictions. One such useful service is high-quality, on-the-fly topography.

Such editing architecture and editor should be recursively embeddable, and be able to respond with reading and writing support for different types of non-program code, including for example string literals, various types of comments, mark-up language tags, and the like. The resultant editor should be able to maintain a linguistically accurate program representation that is updated on every modification, however small, yet should take into account the greater fragility of the representations in the presence of unrestricted textual editing. Such editor should also take into account the potential for confusing behavior and inconsistency between what is seen on a display and what is being represented internally. Thus, there is a need for a preferably programmer-intuitive editor that can provide a representation closely related to displayed text, but that also reflects linguistic structure at all times.

Further, such an editor should be well encapsulable for use with other tools, and preferably would be analogous to a GUI component in not demanding complex support such as a particular kind of source code repository. Finally, since practical software engineering can involve many languages, the editor should be readily configurable via language specifications, and should provide a visual style that is easily configured for different contexts and tasks.

The present invention provides such a language-based editing architecture and embeddable sub-editors.

SUMMARY OF THE INVENTION

The present invention provides a text-oriented tool for editing source code, which tool is fundamentally driven by programming language technology and may be embedded in other graphical user interfaces (GUI). The tool can make language-oriented representations that preferably are configurable by declarative specifications available to other tools. The resultant architecture uses a lexically-oriented internal representation for source code that satisfies constraints associated with usability and with the need to integrate with other tools.

In realtime, each keystroke is examined to identify whether a trigger token has been entered by the programmer using a keyboard, mouse, etc., or perhaps a menu selection. What is a trigger token will depend upon the programing language used during code input by the programmer. Without limitation, exemplary trigger tokens can include ", /*, //, /**, <. A trigger token generally implies the existence of a boundary condition, that a new grammar and syntax should now be invoked.

When a trigger token is identified, a sub-editor appropriate to the specific token is invoked and an appropriate sub-document is created. In the preferred embodiment, the sub-document will automatically be bounded by a pair of closing boundary tokens. The boundary token on the left, e.g., preceding the sub-document, is a subdocument opening token, and boundary token on the right, e.g., following the sub-document, is the subdocument closing token. In the example of a string literal, the opening token and the closing token are both double quotes, although in some cases the boundary tokens may be different from the trigger tokens and even from one another, e.g., < > for HTML tags. On the display, the cursor is automatically placed between the opening and closing boundary tokens. Thus, if a user keystroke is the double quote ", this trigger token is recognized as representing the start of a string literal. The appropriate sub-editor that handles string literals is invoked, and what is seen is on the computer system monitor is " [cursor] ". The programmer then enters whatever is desired for the string literal, the entered keystrokes bounded between the

opening and ending double quote boundary tokens. The programmer then cursors beyond either boundary token, (e.g., beyond a "), whereupon the subdocument (here a string literal) is exited, and control is transparently and seamlessly returned to the parent editor. Sub-editors may also be provided for various types of comments, for mark-up language tags, and program code may include recursively nested sub-documents created by nested sub-editors, each invoked upon recognition of the appropriate triggering token.

The text editor-like look and feel of the present invention transparently handles programmer actions such as manually attempting to input a closing trigger token, and preventing manual deletion of one of a pair of boundary tokens unless the bounded sub-document is empty.

Using a computer system input device such as a keyboard, on a keystroke-by-keystroke basis the present invention examines the potential effect of each keystroke entry or token made by a programmer. Tokens are examined by a parent editor in which the present invention may be embedded to determine whether they are trigger tokens, for example a token such as a double quote " denoting the start of a string literal, a slash asterisk /* denoting the start of a comment, a bracket < denoting the start of a tag in a mark-up language, etc. If the sub-editor determines that the token can or should modify the model or text buffer, the modification is made, and appropriate changes to the view are painted on the screen of a monitor associated with the computer system with which the parent editor and sub-editor(s) are used.

Upon recognizing such model changing trigger tokens, the view is changed to include both opening and closing boundary tokens, and the cursor is placed between such tokens. Thus, if the programmer types a ", the view is changed by the present invention to "[cursor]" and the next keystrokes entered by the programmer will be interpreted as a string literal. The view is consistent with a string literal entry immediately after the first " was entered by the user. This is in stark contrast to prior art editors that do not generally deal intelligently with fragmentary or malformed entered source code, and would present the

programmer with a halted system until the perceived error resulting from a single " was manually detected and corrected. Instead of facing a distracting screen filled with meaningless coloring and fonts resulting from a misinterpreted single " token, the screen will correctly display the string literal contents in the appropriate font, font size, color, indentation, etc.

An editor according to the present invention thus satisfies constraints imposed by a need for enhanced programming services and for tighter integration, and further provides a new architecture based upon a lexical representation of source code. Architecture and systems according to the present invention can be implemented using present graphical toolkits, and can be made highly configurable using present language analysis tools. Further, applicant's architecture may be encapsulated in a manner consistent with re-use in many software engineering contexts. The architecture is different than what has been tried unsuccessfully in the past, but is no more complex.

To recapitulate, when the programmer inputs a single trigger token (perhaps a double quote to denote that a string literal will now be entered), the present invention advantageously provides an appropriate pair of boundary tokens, and locates the cursor on the display intermediate these two boundary tokens. The present invention acts to ensure that the programmer cannot delete only one of the pair of boundary tokens, which would result in an undesired mismatched state. More particularly, the matched pair of boundary tokens cannot be deleted by the programmer unless the entire subdocument defined and surrounded by the two boundary tokens and the contents therebetween is deleted. As a result, the present invention avoids the potentially confusing analyzer and display states due to mismatched programmer-input boundaries, e.g, one " instead of " " noted in the prior art.

Further, the present invention provides one or more embedded sub-editors, wherein a sub-editor is dedicated to the type of contents appearing within the boundaries demarked by an associated pair of boundary tokens. For example,

if the boundary tokens are a pair of double quotes " ", the contents will be a string literal and a sub-editor specially dedicated to dealing with string literal contents is invoked, preferably transparently to the programmer. If the boundary token pair were < and >, the information bounded between these tokens would be tag contents, and a specialized sub-editor dedicated to tag contents would be preferably transparently invoked. Other boundary tokens are possible, for example tokens that precede and follow comments, whereupon a sub-editor dedicated to comment contents would be invoked.

The preferred architecture permits specializing each sub-editor for the purpose at hand, where each sub-editor will use different language rules as to what is legal or not legal for the specific type of contents associated with the sub-editor. Special behavior may be invoked that is appropriate only for an associated content type. There might be a command set appropriate to a cursor located within a string contents, which command set might be inappropriate to other types of contents, e.g., tag contents, or comment contents. Thus the present invention functions with boundaries that are set by boundary tokens, and invokes specialized sub-editors appropriate to the bounded content types, and indeed can invoke recursively embedded sub-editors, depending upon what is being input by the programmer.

Further, the present invention provides a special behavior around boundary tokens to provide a look and feel of a single text editor to the programmer. Thus, having created boundaries demarked by pairs of boundary tokens, the present invention essentially functionally smooths the boundaries to make them less obtrusive to the programmer. However as noted, in reality editing results from a parent editor (that may be a prior art editor) that transparently invokes dedicated sub-editors, included recursively embedded sub-editors as needed, according to the present invention.

Other features and advantages of the invention will appear from the following description in which the preferred embodiments have been set forth in detail in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 depicts a computer system used as a source code editing system, according to the prior art;

FIG. 2A depicts text buffer contents comprising program code and string literal code, according to the prior art;

FIG. 2B depicts text buffer contents intermediate an edit change, showing code sections that will be misinterpreted by prior art editors;

FIG. 3 depicts a computer system used as a source code editing system with embeddable source code editor modules, according to the present invention;

FIG. 4 depicts architecture for a source code processor sub-editor, according to the present invention;

FIGS. 5A-5D depict token replacement, according to the present invention;

FIG. 5E depicts string literal sub-document representation in an editor buffer, according to the present invention; and

FIG. 6 is a screen image of an actual display generated by the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Fig. 3 depicts a system 10' used to edit and display source code, according to the present invention. In some aspects system 10' is similar to prior art system 10. However in contrast, system 10' includes at least one embeddable editor or editor module depicted conceptually as 200, 210 (also referred to as sub-editor(s)), and includes a structural buffer with embedded subdocuments 320', rather than including a prior art text buffer 50 as was shown in Fig. 1. In the preferred embodiment, structural buffer 320' is a token-oriented buffer. Editor modules according to the present invention may be used with prior art program

editors, including editors as disclosed in applicant's incorporated-by-reference U.S. patents 5,752,058 and 5,748,975, specifically the executive editor described therein as element 135.

A programmer enters code using, for example, keyboard 60, or a mouse or trackball to select menu items, or perhaps in an appropriate system using voice commands. As the programmer enters code, the parent editor within computer system 20 functions until an event (e.g., a keystroke, menu selection, or voice-invoked command) representing what is referred to herein as a trigger event or trigger token is encountered. Since a keyboard is commonly used, assume that programmer input is made with a keyboard. In such case, a trigger event or trigger token keystroke denotes that keystroke entries between such trigger events are normally subject to a grammatical specification that is disjoint from the syntax of the rest of the language being input. By way of example and without limitation, trigger event keystrokes may include the following tokens:

"	-	signifies the beginning and the end of a string literal;
/*	-	signifies beginning of multiline comment;
*/	-	signifies end of multiline comment;
//	-	denotes single line comment;
<	-	denotes start of tag in mark-up language;
>	-	denotes end of tag in mark-up language
/**	-	denote document type comment

Fig. 4 depicts architecture and components for a source code editor or sub-editor 290, according to the present invention. Fig. 4 is based upon the Model-View-Controller (MVC) design paradigm, and the horizontal dotted lines show division of the present invention 290 into model (M), view (V), and control (C) functions. The vertical dotted line suggests how editor 290 is separated into core functionality present in every editor (represented by the six modules to the left of the vertical dotted line), and language-specific functionality (represented by the three modules to the right of the vertical dotted line). As such, in constructing an editor or sub-editor specialized for a particular language, one would combine

the editor core (modules 300, 330, 320, 350, 360, 370) with three additional components or modules, each of which was specialized for the language: a language-specific styler 380, a language-specific lexer 340, and a language-specific editor kit 310.

The intent of the division indicated by the vertical dotted line in Fig. 4 is to simplify as much as possible the creation of new language-specific editors, namely by having a reusable core that supports all functionality that is not language specific. Preferably the specific programming technique used to create this division is specialization, a technique known to those skilled in the art of object-oriented programming. By way of example, general purpose code may be encapsulated in a generic class (e.g., AbstractStyler) that by itself would be incapable of the required functionality. In Fig. 4, each language-specific styler 380 would likewise preferably be implemented as a class (e.g., CStyler for the C programming language). Each language-specific styler would then include the code for the abstract styler 370, using the object-oriented programming technique of inheritance. Those skilled in the relevant art will recognize that using the inheritance technique, one class (in this example CStyler) both includes and extends a so-called parent class (in this example AbstractStyler) for a specialized purpose (in this example for C programming language). The result is a styler for the C language (the class named CStyler) that was created only by adding the extra styling code that is specific to that language.

The view functionality portion of editor 290 includes rendering engine 360 and abstract styler 370. Rendering engine creates visual displays on monitor 80 (Fig. 3) based upon two sources of input: a document model 32- (the source code being viewed) and a style description that specifies typographic characteristics such as type face, size, color, background shading, etc. In the case of each token to be displayed, as supplied by the token-based document model, the rendering engine consults the current styler for relevant information. View module 370 represents the styler for the particular language being used. As noted, the styler for each language is procured by combining (through

inheritance or specialization) some generic styling code (module 370) with language-specific styling information (module 380). Thus, what is seen on the monitor screen can differ from the contents of the document model.

Preferably styler modules facilitate a typographically-enhanced display such as shown in Fig. 6 by assigning stylistic properties to each token. The styler module lends itself to being automatically generated, although at present hand-written stylers are used. The stylers can also be used to export plain textual source code by rendering into a character stream, and dropping stylistic information that cannot be represented. Appropriate formatting can be achieved using stylers optimized for text output.

The model functionality portion of editor 290 includes source code model 320 and AbstractLexer 350. Model 320 is a data structure for representing source code as a list of tokens. This data structure need not be specialized for any particular language, and in the preferred embodiment source code model 320 is buffer 320', described with respect to Fig. 3. Lexer 340 is the mechanism whereby ordinary text (either read in from a text file or keyed in by a programmer) is analyzed and converted into the tokens that get stored in the model's data structure. The generic portion of the lexer, AbstractLexer 350, is not specific to any language. Lexer 340, on the other hand, is language-specific and contains rules for forming tokens in the particular language at hand. As noted, there might be a reusable core class named AbstractLexer and a lexer specific to the C language, that extends or specializes, e.g., a lexer named CLexer, which is a particular instance of module 340.

Lexer 340 and AbstractLexer 350 deal with rules that help define the lexemes or lexical classes of the language presently being entered by the programmer using system 10'. Changes deemed necessary to the view are affected by EditorWidget, 300, for example in response to a perceived required change determined by source code model 320 in Fig. 4. AbstractStyler 370 and styler 380 are coupled to a rendering engine 360, that in turn is coupled to source code

model 320. When so required, the rendering engine will obtain the appropriate token text and type, and cause the same to be displayed on the computer monitor associated with the computer system in use. More specifically, rendering engine 360 will cause a display of source code in accordance with the requirements of advanced program typography, with a style update occurring with each keystroke as the entered source code is incrementally re-analyzed.

The controller functionality portion of editor 290 includes EditorWidget module 300 and AbstractEditorKit 330. EditorWidget module 300 is a generic manager for holding the other modules together, and is passed programmer input to computer system 10', for example input entered on keyboard 60, mouse 70, etc. The actual code that gets executed in response to user input is typically implemented by commands supplied by AbstractEditorKit 330, for example code that inserts a new character into a program. EditorWidget 300 dispatches window system events and contributes to making an editor according to the present invention a fully functionally member of the JFC widget family.

EditorKit 310 implements the intricate editing behavior described later herein, and much of the editor kit functionality is language-independent. However some EditorKit functionality may be custom-tuned for each particular language, for example adding keyboard shortcuts for inserting language constructs. Again, EditorKit functionality preferably is implemented in two parts: AbstractEditor kit 330, a generic component not specific to any language, and EditorKit 310, a language-specific component that includes commands specific to a particular language. Thus, a primary responsibility of the EditorKit is to implement user actions that require taking the context of the action into consideration. Some actions such as cursor movement commands require no changes to the source code model, and instead their execution depends only on the context (tokens) surrounding the cursor. Other actions, such as insertions and deletions, may depend not only on the modification context, but also on the state after the modification.

To facilitate this functionality, EditorKit 310 preferably commences a two-stage modification process upon any potential change responsive to a user input. First, source code model 320 is requested via Abstract EditorKit 330 to consider the effects of the change without modifying the underlying content. If a change is deemed to modify the underlying content, source model 320 so advises EditorWidget 300, and an object is produced describing the change in terms of a required model transformation. When the EditorKit regains control, it examines the transformation, and if it is not valid or has no effect the transformation is discarded, otherwise it is applied to the model. As such, the embedded EditorKit and associated modules causes a go/no-go analysis of each keystroke to be carried out, and based upon language rules applicable to what is being entered determine legality of each entry. This is in contrast to prior art pattern matching that upon discerning what appears to be an improper pattern simply advise the programmer than an error has occurred, and typically produce a view that is of no help to the programmer in understanding what perceived error is being referenced.

To recapitulate what is shown in Fig. 4, the architecture of editor 290 is advantageously decomposed along two different dimensions: functionality (MVC) and core versus language-specific code. The result is that it is not necessary to reimplement a whole editor to achieve the design goal of having different editors for different contexts. Thus, the editor that specializes in general Java code is different from the editor that specializes in the contents of strings, but not completely different. All of the core functionality may be reused, and it is only necessary that language-specific details differ. The described MVC architecture is advantageous for display and editing, and generally reflects the design of the Java™ Foundation Classes (JFC) "Swing" toolkit. Although the described partition of functionality was achieved by subclassing (also known as inheritance or as specialization), other decompositions may instead be used. In the above-described MVC paradigm, an exemplary controller may be such as the so-called keystroke executive (element 130) described in applicant's U.S. patents 5,748,975 and 5,752,058.

Referring to Fig. 3 and Fig. 6, the view might cause comments to appear in one color and font type, and cause source code to be displayed in another font, color, and perhaps with different indentation, etc. Although shown in monochrome, Fig. 6 depicts a typical screen image in which different code types are differently displayed, according to the present invention. For example, in the actual display, the terms "import" may be displayed with a brown font, "public" with an orange font, "class" with a green front, "StringExample" with a purple font, and so forth. The ability of the present invention to respond intelligently to reading and writing comments is in contrast to prior art editors, to which comments are often simply transparent, and tedious to format, despite their importance to readers. The present invention specializes tasks that confront programmers and can provide additional or enhanced editing services, e.g., automatic indentation of source code lines involving services based loosely on linguistic structure. The resultant forms helps reading by providing visual feedback for nesting, and helps code writing by saving tedious keystrokes.

Consider Figs. 5A-5C in which an exemplary method of token replacement is depicted. Fig. 5A depicts the dynamic state of the model or preferably token oriented buffer 320', showing four tokens A, +, C, and ; with the cursor initially between the + and C tokens. It is understood that the view seen as display 90 on monitor 80 would not generally show the "boxes" depicts in Fig. 5A to demark or delineate each token. Assume that the programmer wishes to insert the two characters =X at the location of cursor 100 in Fig. 5A. Fig. 5B depicts an intermediate step in which the controller (or keystroke executive 130 depicted in applicant's U.S. patents 5,748,975 and 5,752,058) consults the lexer, the lexer producing new tokens += and XC, as shown in Fig. 5B. These new tokens replace middle tokens + and C in Fig. 5A. Fig. 5C depicts the content after the desired transformation has been applied, namely showing four tokens A and += and XC and ;.

In the preferred implementation, when a programmer enters a trigger token, the present invention will be invoked from the parent editor, e.g., perhaps an editor as exemplified in applicant's cited patents, although other parent editors may instead be used. Thus in Fig. 5D, as soon as the trigger token " is recognized, the editor widget 300 will recognize that the source code model should now be changed to reflect that a string literal is about to be entered. As shown in Fig. 5E, the present invention will automatically insert opening and closing boundary tokens " ", and will cause the model and the display to be as shown, with the cursor between the two boundary tokens, e.g., " [cursor] ". On display 90, the string-literal entries bracketed by the " and " can be painted in a different font with a different font size, different color, etc. to ease the programmer's view of what is being entered. However the confusing display encountered with prior art editors once the initial token, here ", has been entered is avoided.

Fig. 5F is an exemplary representation of a string literal in an edit buffer, according to the present invention, wherein the string is treated as a subdocument. Assume the programmer wishes to enter the following source code line:

S = E + "ABCD";

As to the group of tokens 117, namely S + E + no decision need be made as to how the rest of the source code is represented. In the example of Fig. 5F, a lexical token representation is made rather than a text buffer representation, as would be typical of the prior art.

As the programmer types the opening quote 119, the " is recognized as a special trigger token. Although it may appear to be an ordinary quote mark, in reality it has a special behavior, as will the closing boundary token, quote 123, which is created automatically by the present invention.

Between the two quotes119, 123 is a string 121 whose contents are represented as a special object that acts as a wrapper for a separate document that holds the

contents of the string, namely a sub-document 127 that holds contents ABCD.

Thus, what will appear on computer monitor 80 for viewing by the programmer will be:

S = E + "AB[cursor]CD";

As soon as the cursor, denoted [cursor], is moved within the pair of quotes, a separate sub-editor, according to the present invention, is invoked to handle the editing activity. As soon as the cursor is moved by the programmer outside either quote, editing activity returns to the parent editor, which may be a prior art editor.

Thus, once the programmer has completed entering the string literal, cursoring beyond either token " will exit the sub-editor and sub-editor document, and return control to the parent editor. As far as the programmer is concerned, the transfer into and out of the sub-document (produced by the sub-editor) is seamless and transparent, and has the look and feel of an ordinary word processor. Use of the invention involves skills no more complicated than those used by users familiar with word processers.

It is possible that from habit the programmer might enter the closing trigger token, e.g., a closing double quote in the example of a string literal. In such case, optionally the sub-editor will respond to such programmer input by moving the cursor over the closing boundary token (e.g., the closing double quote that was automatically inserted by the sub-editor), and by returning control to the parent editor. This optional feature helps make the present invention more transparent and maintains a desired look and feel. Care is taken, however, to ensure that this optional feature does not occur if a programmer types an escaped-trigger character, e.g., \".

The present invention ensures that the programmer cannot manually delete one of pair of boundary tokens, e.g., an opening or closing ". But if the sub-document

bounded by the boundary tokens is empty, manual deletion of one boundary token by the programmer is caused to delete both boundary tokens as well as the entire sub-document. Optionally a textual look and feel can be somewhat preserved when the user attempts to delete a boundary token bounding a non-empty string or sub-document by simply moving the cursor over rather than deleting the token in question.

Thus, when the programmer wishes to delete the entire string literal, the preferred embodiment requires multiple keystrokes, e.g., placing the cursor to the right of the closing boundary token " and back-spacing several times, or placing the cursor to the left of the opening token " and pressing the delete key several times. Alternatively as suggested by Fig. 3, a pull-down menu can be invoked to erase highlighted text, e.g., the string literal. Indeed, if desired a pull-down menu rather than a keyboard could be used to input a trigger token, whereupon the present invention would provide an appropriate pair of boundary tokens and place the cursor between the boundary tokens. If the cursor is to the right of the closing boundary token, the backspace key must be pressed a sufficient number of times to delete the entire contents surrounded by the boundary token pair, and then pressed again to delete the opening boundary token, whereupon the closing boundary token is automatically deleted. At this juncture the sub-editor (or child editor) has completed its immediate task and the parent editor is again in charge. If deleting from the left of the opening boundary token, the same procedure is followed, except with the delete key rather than with the backspace key. The preferred embodiment can further permit blocking the boundary token surrounded contents and then continuing the block to include one of the boundary tokens. The blocked contents and boundary token can then be deleted, whereupon the remaining boundary token is also automatically deleted.

While the above example dealt with distinguishing and viewing string literals as contrasted with program code during programmer entry, the present invention can provide appropriate sub-editors or child editors in response to recognition of

other triggering tokens, e.g., < for mark-up language entries, /*, //, /** and so forth. In each instance, once the programmer's enters the opening such keystroke, recognition of the trigger token invokes the sub-editor appropriate thereto. The model is changed to show opening and closing boundary token, and the cursor is placed between two tokens on the viewing screen. Thus, if the programmer had entered <, the mark-up language sub-editor is invoked, a mark-up subdocument is opened, and the model will reflect < [cursor]>. As noted, the font, font color, etc. for the tag entry between the < and > tokens can be made different than non-tag entries for ease of viewing. Similarly when the programmer enters the trigger token /*, the present invention will cause the model and view to appear as /* [cursor] */. The programmer's comments will automatically appear between the opening and closing comment boundary tokens, and may be painted in a font, font size, color, with indentation, etc. appropriate for comment entries.

As noted, it is very possible that embedded sub-editors can be invoked within embedded sub-editors. Table 1 and Table 2 below provide two exemplary Java™ technology interfaces for the present invention. These interfaces depict how editor instances communicate with one another to coordinate responses to ordinary editing commands near the boundary between an outer or parent editor, and an embedded or child editor instance, according to the present invention, the child editor (or sub-editor) being embedded within the parent editor. The exemplary interfaces enable programmer visible behavior that smooths over parent-child editor boundaries. Tables 1 and 2 are exemplary in that other interfaces enable the present invention to smoothly and transparently traverse editor boundaries for languages and technology other than Java™ technology.

The parent-child interface relationship for editor embedding reflects document-subdocument relationships in the document being edited. Preferably when an editor instance for an outer document is created, which enables the programmer to view and possibly to modify that document, there is also created a child editor instance for each subdocument of the main document. Each child editor

instance will be of a kind appropriate to the particular situation and will depend upon factors such as the type of subdocument, the general task context in which the programmer is working, and the personal preferences of the programmer.

Situations can occur in which a particular editor instance might play both roles: as a child within an outer editor instance, but with other editor instances embedded within it. For example this situation would occur where documents become nested more than one document deep. It will be appreciated the nested subdocuments might be of different types, e.g., comments, tags, etc. In some applications additional interfaces could be provided to support additional coordination between parent and child editors, above and beyond routine editing operations.

TABLE 1: PARENT EDITOR - CHILD EDITOR INTERFACE

```
/*
  * An interface to be implemented by any editor that will have at least one child
  * editor instance embedded within, which is to say, any editor that will play the
  * role of parent in an embedding relationship among editor instances. These
  * methods permit the parent to be called by the child editor when needed to
  * respond to some programmer input.
  */
public interface ParentEditor {
        /**
         * Instructs the editor to acquire keyboard input focus from the window
         * system and to position the cursor at the boundary of a child editor. This
         * allows a child to "move" the cursor over the boundary from child into
         * parent.
         *
         * @param child  The child editor at whose boundary the cursor should
         * be positioned.
         *
         * @param before Specifies at which side of the child editor the cursor
         * should be positioned: before if true, after if false.
         **/
public void takeFocus (ChildEditor child, boolean before);

        /**
         * Instructs the editor to delete a subdocument of the document
         * being viewed, as well as the child editor associated with the
         * subdocument. This permits editing operations initiated from
         * within a child editor to have the effect of deleting the whole
         * subdocument, not just the subdocument's contents (which could
         * be accomplished purely within the child editor).
```

```
         *
         * @param child The child editor whose subdocument is to be
         * deleted.
         */
public void deleteChild(ChildEditor child);
}
```

TABLE 2: EDITOR-EDITOR AS CHILD EDITOR INTERFACE

```
/**
 * An interface to be implemented by any editor whose instances may be
 * embedded within another editor instance, which is to say: any editor that
 * will play the role of child in an embedding relationship. These
 * methods permit the child to be called by the parent editor when needed to
 * respond to some programmer input. */
public interface ChildEditor {

        /**
         * Allows a parent editor to determine whether the subdocument
         * associated with a child has any content at all.  Some editing actions
         * in the parent might, in the case of child editor whose contents are
         * empty, lead to the deletion of the subdocument and its corresponding
         * child editor.
         *
         * @return true if the associated subdocument is empty
         */
        public boolean isEmpty();

        /**
         * Allows a parent editor to determine whether an embedded editor
         * is prepared to acquire keyboard input focus.  If not, a parent
         * editor may well chose to interpret navigation commands at the
         * child's boundary so that they just pass over the particular sub
         * document/editor with no interaction.
         *
         * @return true if the editor can and is prepared to acquire keyboard
         * input focus.
        **/
        public boolean canTakeFocus();

        /**
         * Instructs the editor to acquirer keyboard input focus from the
         * window system and to position the cursor within the editor.  This
         * allows a parent to "move" the cursor over the boundary from parent
         * into a child.
         *
         * @parem offset The character position at which the cursor should be
         * placed.  If the specified position is negative, then position the cursor
         * at the extreme right position within the subdocument.
```

```
     **/
     public void takeFocus (int offset);

}
```

Turning now to the model component of Fig. 4, as has been seen, the present invention represents entered source code as a sequence of lexical tokens. This representation preferably is extended in several ways to promote flexibility in supporting the required user-model and in fitting naturally with the incremental lexical analysis algorithm, stored in or loadable into memory 40 and executable by CPU 30. The lexical analysis and algorithm preferably is fully general and supports unbounded contextual dependency and multiple lexical states. Moreover, incrementality can be crafted onto existing batch lexers that conform to a simple interface. For instance, in the preferred embodiment, for the Java programming language lexer 340 is generated by the JavaCC tool from a readily available lexical specification that is extended to include various categories of irregular lexemes created during editing, as discussed herein.

Fig. 6 is a reproduction of an actual display 90 as viewed by a programmer on a monitor 80 (see Fig. 3). Advantageously embeddable editors according to the present invention can provide advanced typographic styles that are dynamically updated with every keystroke, as the source code being entered is incrementally re-analyzed. Alternative styles for each language may be selected dynamically, for example to suit individual programmer preference, or perhaps to accommodate particular tools driving the display. The style shown in Fig. 6 is configured by many token categories to which many separate token styles are assigned. As a practical matter much of the stylistic detail compensates for the absence of type faces suitable for displaying programming. Each token style specifies type face, size relative to a base size, style (plain, bold, italic), foreground and background colors, baseline elevation, left and right boundary specifications used to compute display spacing between adjacent tokens. Token styles may also specify alternative display glyphs, for example when necessary to display ligatures.

Styles used with the present invention reveal that certain tokens may be lexically incomplete (for example "0x") or may be badly formed (for example "08"), based upon lexical grammars extended to include such tokens. Nonetheless the present invention treats such tokens as legitimate in every other respect. A surprising amount of visual detail may be achieved using only lexical information. Indentation requires fuzzy parsing in the style of many text editors. Additional visual features could be added through other kinds of fuzzy parsing, for example adjusting operating spacing based upon expression depth. In the preferred embodiment, horizontal spacing between tokens is computed from the source code and is not affected by pressed of the keyboard space bar. This feature is found to improve legibility and to save keystrokes. If desired, a tab-like mechanism could be provided to permit programmers to impose some control over vertical alignment.

In most respects, the present invention behaves like a code-oriented text editor, and where differences exist, the editor is made to appear to do the right thing when used as a text editor. Some behaviors are completely conventional, for example, indentation is automatic, and line breaks are explicitly entered and deleted by the programmer. The preferred embodiment does not presently break lines but a linguistically driven mechanism could be provided to wrap lines wider than the available window. Typing within comment tokens and language tokens (especially string literal tokens) is conventional, with the notable exception that the programmer can readily type multi-line comments and, with some modification, strings, as shown in Fig. 6.

Non-standard behavior appears in and around token boundaries, which to a first approximation are determined purely by the lexical analyzer 340, 350. When the cursor rests between two tokens, it is displayed midway between and pressing the space bar has no effect. As noted, where necessary to unambiguously compute a boundary, source code model 320 will insert a separator token, e.g., between keywords. Advantageously the source code model enables stable

references to source code structure. From the perspective of tool integration this is desired in that the identity of unaffected tokens is guaranteed.

Optionally, source code model 320 may be made responsible for adding and removing "separators", which are special non-linguistic tokens automatically inserted between keywords to demark boundaries that might not otherwise be unambiguously computable. Inserting a separator token between adjacent lexical tokens is somewhat like inserting a "smart space" with a word processor, and the cursor may rest on either side of a separator. Deleting a separator is treated as a request to join surrounding lexical tokens. Note that if the surrounding lexical tokens could not have been joined, there would presently be no separator between these tokens. Separators may come and go as the lexical categories of adjacent tokens are changed during editing. However since separator tokens are behavioral rather than visual, their coming and going is not distracting to the programmer.

The use of embedded editors will now be further described. It is noted that the contents of string literals obey different grammars than the surrounding code, and thus receive special treatment according to the present invention. By contrast, the contents of comments are not analyzed at all. The surrounding code versus string literal regions are specially treated, beginning with behavior that preserves their boundaries during normal editing. Although somewhat akin to structure editing, such treatment solves many traditional problems associated in the prior art with boundary confusion. Having guaranteed boundary stability for these regions, the present invention provides specialized behavior in a straightforward fashion: specialized editors are simply embedded to match the model. One type of embedded editor is for strings, another is for character literals, yet another for plain text comments. Other such embedded editors can support mark-up language (e.g., HTML, SML), or graphical comments. The resultant architecture is specialized for source code and designed such that boundaries are no more obtrusive than absolutely necessary. For example, the

programmer can move the text cursor on the display smoothly across boundaries, between code and embedded structures.

The representation of embedded structures will now be described with further reference to Fig. 4. Programming languages commonly include embedded syntactic structures that have distinct lexical rules, most notably comments and strings. As noted above, the present invention supports embedded structures using nested editors that have transparent boundaries. The underlying requirement for support is readily met with the relevant embedded language structures addressed herein in that the structures have well defined linguistic boundaries. Further, their contents can be tokenized as a single entity by a language lexer, for example lexer 340, 350 in Fig. 4.

The architecture of Fig. 4 advantageously permits using any editors in the JFC text framework, including using the present invention recursively. Mapping from token types to editor types preferably is performed by the language module. In the preferred embodiment, this module uses a standard JFC text editor for comments, and a token-based editor, according to the present invention, for strings and for character literals, although other editor types could also be used. Both strings and character constants afford a simple lexical description that recognizes character escapes such as \n, \t, etc. This permits highlighting legal escapes, such that these character escapes are distinguishable from the rest of the text. Further, character escapes permit indicating valid versus invalid escapes.

It will now be appreciated that the present invention differs from prior art editors, especially with respect to cause and effect. Lexical representations are used in a stream rather than in a tree, and thus bear a closer relationship to textual source code. Analysis needed to update the representation after each keystroke edit usually requires only local context, and is suitable for program fragments. Sufficient linguistic information is available to provide many language-based services, including more robust implementation of familiar services such as

indentation, parenthesis, and bracket matching, procedure or method head recognition, etc. The resultant language representation is suitable for integration with other tools including complete language analyzers. If desired, further analysis including parsing could be folded into the invention, but at increased cost and complexity.

As has been described, the resultant textual display and behavior has a sufficiently familiar look and feel as to require essentially no training to master. Further, the display can be specialized for use with programs using only lexical information, and a fine-grained typographical display is implementable and configurable with present toolkit technology. Lexical token representation is sufficiently robust, even in the presence of partially typed and badly formed tokens, including management of string literals when a double quote (or bracket) is missing. As described, preferably many sub-editors are provided, including sub-editors to handle comments and possibly other non-textual, annotations.

In summary, it is seen that the present invention meets the goals that have been elusive in the prior art. Use of the invention is visually rich but otherwise unobtrusive. Nearly all keystroke sequences to users of word processors still have their intended effect, with the added bonus of fine-grained visual feedback. The programmer is now aided rather than distracted by the displayed information, and is free to concentrate more completely on the task at hand, namely understanding and writing source code. Further, the rich display engine provided presents opportunities for tools to present information by modulating the source code display to suit the task at hand. Although the preferred embodiment advantageously provides sub-editors that are embeddable, an editor architecture could be provided in which a relatively complex master editor was used without embeddable sub-editors. However, the use sub-editors that are embeddable, according to the present invention, advantageously greatly simplifies the overall system design and more particularly simplifies the design of each specialized sub-editor.

Modifications and variations may be made to the disclosed embodiments without departing from the subject and spirit of the invention as defined by the following claims.